May-Happen-in-Parallel Analysis with Condition Synchronization

Elvira Albert[1], Antonio Flores-Montoya[2], and Samir Genaim[1]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany

Abstract. Concurrent programs can synchronize by means of conditions and/or message passing. In the former, processes communicate and synchronize by means of shared variables that several processes can read and write. In the latter, communication is by sending, receiving and waiting for messages. Condition synchronization is often more efficient but also more difficult to analyze and reason about. In this paper, we leverage an existing *may-happen-in-parallel* (MHP) analysis, which was designed for a particular form of message passing based on future variables, to handle condition synchronization effectively, thus enabling the analysis of programs that use both mechanisms. The information inferred by an MHP has been proven to be essential to infer both safety properties (e.g., deadlock freedom) and liveness properties (termination and resource boundedness) of concurrent programs.

1 Introduction

With the trend of parallel systems and the emergence of multi-core computing, the development of techniques and tools that help analyzing and verifying the behaviour of concurrent programs has become fundamental. Concurrent programs contain several processes (or tasks) that work together to perform a task. For that purpose, they communicate and synchronize with each other. Communication can be programmed using conditions or using *future variables* [7].

In order to develop our analysis, we consider a generic asynchronous language in which tasks can execute across different task-buffers in parallel; and inside each buffer, tasks can interleave their computations. The language allows both synchronization using conditions and future variables. When future variables are used for synchronization, one process notifies through the future variable that its execution is completed and the notification is received by the process(es) waiting for its completion. In particular, the instruction `Fut f=b!m();` posts an asynchronous task `m` on buffer `b` and allows the current task to synchronize with the completion of `m` by means of the future variable `f`. The instruction **await** `f?` is used to synchronize with the completion of `m` as follows. If the process executing `m` has not finished when executing the **await**, the future `f` is not ready, and the current process is suspended. In such case, the processor can be released such that another pending process in the same buffer can take it and start to execute (thus interleaving its computation with the suspended task). When condition synchronization is used, one process writes into a variable that is read by another. Thus, instead of using future variables, the instruction **await** takes the form **await** `b?` where `b` is a Boolean condition involving shared variables. For instance, we can write **await** `x!=`**null** that synchronizes on the condition that the shared variable `x` is not **null**. The use of condition synchronization is known to pose challenges in static analysis. This is because it is difficult to automatically infer to which parts of the program the **await** instruction synchronizes. As a consequence, condition synchronization is more difficult to debug and analyze, while its main advantage is efficiency – it lacks the overhead of managing future variables. In contrast, future variable based synchronization is less efficient but it helps in producing concurrent applications in a less error-prone way as it is clearer how processes synchronize.

May-happen-in-parallel (MHP) is an analysis which identifies pairs of statements that can execute in parallel across several buffers and in an interleaved way within a buffer (see [13, 3]). MHP directly allows ensuring absence of data races in the access to the shared memory. Besides, it is a crucial analysis to later prove more complex properties like termination, resource consumption and deadlock freedom. In [15, 9], MHP pairs are used to greatly improve the accuracy of deadlock analysis. The language we consider uses the instruction `f.`**get** to block the execution of the current task until the task associated with the future `f` has finished. Consider method "`void m (Buffer b) {Fut f=b!foo(); f.get;}`". Given two buffers b_1 and b_2, if we execute the processes `m(`b_2`)` on b_1 and `m(`b_1`)` on b_2 in parallel, we can have a deadlock situation as they block waiting for each other to terminate. The MHP analysis allows discarding infeasible deadlocks when the instructions involved in a possible deadlock cycle cannot happen in parallel. For instance, if we are sure that the execution of `m(`b_2`)` in b_1 has finished before the execution of `m(`b_1`)` in b_2 starts (i.e., the tasks cannot happen in parallel), we can prove deadlock freedom. Also, MHP improves the accuracy of termination and cost analysis [4] since it allows discarding infeasible interleavings. For instance, consider a loop like "`while (l!=null) {x=b!p(l.data); await x?; l=l.next;}`", where the instruction `await x?` synchronizes with the completion of the asynchronous task to `p`. If the asynchronous task is not completed (`x` is not ready), the current task releases the processor and another task can take it. This loop terminates and has a bounded resource consumption provided no instruction that increases the length of the list `l` *interleaves* or *executes in parallel* with the body of this loop.

In this paper, we leverage an existing MHP analysis [13, 3] developed for synchronization using future variables to handle condition synchronization effectively. Handling both future variables and shared memory synchronization is difficult because the analysis has to infer soundly the program points which the **await** instructions synchronize with and propagate them accordingly to both kinds of synchronization. Our analysis is based on the *must-have-finished* (MHF) property which allows us to determine that a given instruction will not be executed afterwards after a certain point. In particular, we aim at inferring MHF-sets which are the set of program points that MHF when the execution reaches a program point of interest. Developing our extension for condition synchronization amounts to developing an analysis that infers the required MHF-sets and using them to refine the original MHP analysis.

2 Language

We consider asynchronous programs with multiple task buffers (see [8]) which may concurrently interleave their computations. The concept of task buffer is materialized by means of an *object*. Tasks from different objects (i.e., different task buffers) execute in parallel. Tasks can be synchronized with the completion of other tasks (from the same or a different buffer) using futures and conditions on shared variables. The number of task buffers does not have to be known a priori and task buffers can be dynamically created. Our model captures the essence of the concurrency and distribution models used in actor-languages (including concurrent objects [12], Erlang [1] and Scala [11]) and in X10 [13], which rely on futures and message passing synchronization. It also has many similarities with the concurrency model in [8] which uses condition synchronization.

2.1 Syntax

A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. One of the methods, named main, corresponds to the initial method which is never posted or called and it is executing in an object with identifier 0. The grammar below describes the syntax of our programs. Here, T are types, m method names, e expressions, x can be field accesses or local variables, b is a Boolean condition involving fields, and y are future variables.

$$\begin{array}{ll} CL ::= class\ C\ \{\bar{T}\ \bar{f}; \bar{M}\} & M ::= T\ m(\bar{T}\ \bar{x})\{s; \textbf{return}\ e;\} \\ s\ \ ::= s; s \mid x = e \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \mid & \\ \qquad \textbf{await}\ y? \mid \textbf{await}\ b \mid x = \textbf{new}\ C(\bar{e}) \mid y = x!m(\bar{e}) \mid y.\textbf{get} & \end{array}$$

The notation $\bar{T}\ \bar{f}$ is used as a shorthand for the sequence $T_1\ f_1; \ldots; T_n\ f_n$, where T_i is a type and f_i is a field name. We use the special identifier *this* to denote the current object. For the sake of generality, the syntax of expressions and conditions is left free and also the set of types is not specified. We assume that every method ends with a **return**.

Each object represents a task-buffer and has a heap with the values assigned to its fields. The concurrency model is as follows. Each object has a lock that is shared by all tasks that belong to the object. Data synchronization is by means of future variables and conditions as follows. An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}b!m(\bar{z})$ such that **await** $y?$ is executed only when the future variable y is available (and hence the task executing m on object b is finished). In the meantime, the object's lock can be released and some *pending* task on that object can take it. The synchronization instruction **await** b blocks the execution until the Boolean condition b evaluates to *true*, and allows other tasks to execute in the meantime. The instruction $y.$**get** blocks the object (no other task of the same object can run) until y is available, i.e., the execution of $m(\bar{z})$ on b is *finished*. Note that the difference from **await** $y?$ is in that is *blocks* the object.

$$(\textsc{newobject})\quad \frac{\text{fresh}(\mathit{oid}')\ ,\ l' = l[x \rightarrow \mathit{oid}'],\ t = \mathit{tsk}(\mathit{tid}, m, l, \langle x = \textsf{new}\ C(\bar{e}); s\rangle), f' = \mathit{init\_atts}(C, \bar{e})}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{t\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l', s)\} \cup \mathcal{Q}) \parallel \mathit{obj}(\mathit{oid}', f', \bot, \{\}) \parallel B\end{array}}$$

$$(\textsc{select})\quad \frac{t = \mathit{tsk}(\mathit{tid}, \_, \_, s) \in \mathcal{Q},\ s \neq \epsilon(v)}{\mathit{obj}(\mathit{oid}, f, \bot, \mathcal{Q}) \parallel B \rightsquigarrow \mathit{obj}(\mathit{oid}, f, \mathit{tid}, \mathcal{Q}) \parallel B}$$

$$(\textsc{async})\quad \frac{l(x) = \mathit{oid}_1,\ \text{fresh}(\mathit{tid}_1),\ l' = l[y \rightarrow \mathit{tid}_1],\ l_1 = \mathit{buildLocals}(\bar{z}, m_1)}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle y = x!m_1(\bar{z}); s\rangle\} \cup \mathcal{Q}) \parallel \mathit{obj}(\mathit{oid}_1, f_1, \_, \mathcal{Q}') \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l', s)\} \cup \mathcal{Q}) \parallel \\ \mathit{obj}(\mathit{oid}_1, f_1, \_, \{\mathit{tsk}(\mathit{tid}_1, m_1, l_1, \mathit{body}(m_1))\} \cup \mathcal{Q}') \parallel B\end{array}}$$

$$(\textsc{await1})\quad \frac{c = y?,\ l(y) = \mathit{tid}_1,\ \mathit{tsk}(\mathit{tid}_1, \_, \_, s_1) \in \mathcal{Q}, s_1 = \epsilon(v) \vee c = b,\ \mathit{eval}(b, f, l) = \mathit{true}}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle \textsf{await}\ c; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{await2})\quad \frac{c = y?,\ l(y) = \mathit{tid}_1,\ \mathit{tsk}(\mathit{tid}_1, \_, \_, s_1) \in \mathcal{Q}, s_1 \neq \epsilon(v) \vee c = b,\ \mathit{eval}(b, f, l) = \mathit{false}}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle \textsf{await}\ c; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \bot, \{\mathit{tsk}(\mathit{tid}, m, l, \langle \textsf{await}\ c; s\rangle)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{get})\quad \frac{l(y) = \mathit{tid}_1,\ \mathit{tsk}(\mathit{tid}_1, \_, \_, s_1) \in \texttt{Obj}, s_1 = \epsilon(v), l' = l[x \rightarrow v]}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, \mathit{oid}, l, \langle x{=}y.\textsf{get}; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, \mathit{oid}, l', s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$(\textsc{return})\quad \frac{v = l(x)}{\begin{array}{c}\mathit{obj}(\mathit{oid}, f, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle \textsf{return}\ x; \rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ \mathit{obj}(\mathit{oid}, f, \bot, \{\mathit{tsk}(\mathit{tid}, m, l, \epsilon(v))\} \cup \mathcal{Q}) \parallel B\end{array}}$$

Fig. 1. Summarized Semantics

Note that our concurrency model is *cooperative* as processor release points are explicit in the code, in contrast to a *preemptive* model in which a higher priority task can interrupt the execution of a lower priority task at any point. Without loss of generality, we assume that all methods in a program have different names.

2.2 Semantics

A *program state* $St = \texttt{Obj}$ is of the form $\texttt{Obj} \equiv \mathit{obj}_1 \parallel \ldots \parallel \mathit{obj}_n$ denoting the parallel execution of the created objects. Each *object* is a term $\mathit{obj}(\mathit{oid}, f, \mathit{lktid}, \mathcal{Q})$ where oid is the object identifier, f is a mapping from the object fields to their values, lktid is the identifier of the *active task* that holds the object's lock or $\bot$ if the object's lock is free, and $\mathcal{Q}$ is the set of tasks in the object. Only one task can be *active* (running) in each object and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $\mathit{tsk}(\mathit{tid}, m, l, s)$ where tid is a unique task identifier, m is the method name executing in the task, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state.

The execution of a program starts from an initial state where we have an initial object with identifier 0 which has no fields and is executing task 0 of the form $S_0 = \mathit{obj}(0, [], 0, \{\mathit{tsk}(0, \textsf{main}, l, \mathit{body}(\textsf{main}))\})$. Here, l maps parameters to their initial values and local reference and future variables to **null** (standard initialization), and $body(m)$ refers to the sequence of instructions in the method m. The execution proceeds from S_0 by selecting *non-deterministically* one of the objects and applying the semantic rules depicted in Fig. 1. We omit the treatment of the sequential instructions as it is standard.

NEWOBJECT: an active task tid in object oid creates an object oid' of type C, its fields are initialized (init_atts) and oid' is introduced to the state with a free lock. SELECT: this rule selects non-deterministically one of the tasks that is in queue and is not finished, and it obtains its object's lock. ASYNC: A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is associated to the corresponding future variable y in l'. We have assumed that $oid \neq oid_1$, but the case $oid = oid_1$ is analogous, the new task tid_1 is simply added to $\mathcal{Q}$ of oid. AWAIT1: It deals with synchronization both on future variables and shared memory. If the future variable we are awaiting for points to a finished task or the condition evaluates to **true** (we use function *eval* to evaluate the condition), the **await** can be completed. When using future variables, the finished task t_1 is looked up in all objects in the current state (denoted `Obj`). Similarly, the evaluation of the condition will require accessing object fields and possibly local variables (thus f and l are looked up). AWAIT2: Otherwise, the task yields the lock so that any other task of the same object can take it. GET: It waits for the future variable but without yielding the lock. Then, it retrieves the value associated with the future variable. RETURN: When **return** is executed, the return value is stored in v so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$) but it does not disappear from the state as its return value may be needed later.

Example 1. Our motivating example is showed in Fig. 2. It consists of two classes A and C and a main method from which the execution starts. The main method receives two input parameters. For simplicity class C and return instructions have been omitted. We will see later that a naïve analysis of the synchronization instructions will report termination and resource boundness problems that are spurious (i.e., false alarm). The trace $S_0 \rightsquigarrow S_1 \rightsquigarrow^* S_2 \rightsquigarrow^* S_3 \rightsquigarrow S_4$ where

$$
\begin{array}{rl}
S_0 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0 \equiv [m \mapsto m_0, n \mapsto n_0],\tilde{2})\}) \\
S_1 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0,\ \tilde{3})\}) \parallel obj(1,f_1,\bot,\{\}) \\
S_2 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0,\tilde{5})\} \\
& \parallel obj(1,f_1,\bot,\{tsk(1,\ \mathsf{init}\ ,l_1,\ \tilde{1}1),\{tsk(2,\mathsf{go},l_2,\ \tilde{2}5)\}) \\
S_3 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0,\tilde{5})\} \\
& \parallel obj(1,f_1,2,\{tsk(1,\ \mathsf{init}\ ,l_1,\ \tilde{1}1),\{tsk(2,\mathsf{go},l_2,\ \tilde{2}5)\}) \\
S_4 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0,\tilde{5})\} \\
& \parallel obj(1,f_1,\bot,\{tsk(1,\ \mathsf{init}\ ,l_1,\ \tilde{1}1),\{tsk(2,\mathsf{go},l_2,\ \tilde{2}5)\}) \\
S_5 \equiv & obj(0,[],0,\{tsk(0,\mathsf{main},l_0,\tilde{5})\} \\
& \parallel obj(1,f_1,1,\{tsk(1,\ \mathsf{init}\ ,l_1,\ \tilde{1}1),\{tsk(2,\mathsf{go},l_2,\ \tilde{2}5)\})
\end{array}
$$

```
1 main (int m, int n) {
2   A a = new A(m,n);
3   a!init();
4   a!go();
5 }
6
7 class A <int g, int f>{
8   C x = null;
9
10   void init() {
11    while (g>0) {
12        Fut y = incrf();
13        await y?
14        g--;
15     }
16    x = new C();
17   }
18   void incrf() {
19     f=2*f;
20   }
21   void incrg() {
22     g=2*g;
23   }
24   void go() {
25     await x!=null;
26     while (f>0) {
27          Fut z = incrg();
28          await z?
29          f--;
30       }
31   }
32 }
33
```

Fig. 2. Synchronization using Boolean conditions.

corresponds to few execution steps starting from method main and two constant values m_0 and n_0 as input parameters. (1) S_0 is the initial state, it includes one object with the main method whose next instruction to be executed is the one at program point 2 (denoted as $\tilde{2}$) and the local variable mapping l_0 keeps the bindings for the input arguments; (2) S_1 is obtained from S_0 by executing the first instruction of method main, using rule NEWOBJECT, that creates object 1 and initializes the two class fields f and g with the values that are passed as arguments in the **new** instruction (the bindings for the fields are kept in f_1); (3) S_2 is obtained from S_1 by applying rule ASYNC twice, to execute program points 3 and 4 of the main method to create the tasks 1 and 2 and add them to the queue of object 1; (4) S_3 is obtained from S_2 by applying rule SELECT that selects task 2 from object 1 to start to execute (5) S_4 is obtained from S_3 by applying rule AWAIT2 that evaluates the boolean condition of the **await** at 25 to **false** and thus releases the processor. Note that this condition synchronizes with the instruction at line 16 which creates the object x and thus allows task 2 to move forward; (6) at S_5 task 1 is selected for execution at object 1 and its execution can proceed till the loop finishes iterating and afterwards object x is created. Up to that point, the task 2 is blocked in the **await** instruction.

3 MHP: Concrete Definition and Static Analysis

In this section, we define the property may-happen-in-parallel, and summarize the main points of the analysis of [3] which over-approximates this property. Finally, we describe a straightforward extension of this analysis to handle condition synchronization which is simple but imprecise.

3.1 Concrete Definition

We first define the concrete property "MHP" that we want to approximate using static analysis. In what follows, we assume that instructions are labelled such that it is possible to obtain the corresponding program point identifiers. Given a sequence of instructions s, we use $pp(s)$ to refer to the program point identifier associated with its first instruction, and $pp(\epsilon(v))$ refers to the exit program point.

Definition 1. *A program point p is active in a state $S = \mathtt{Obj}$ within task tid, iff there is $obj(oid, \_, \_, \mathcal{Q}) \in \mathtt{Obj}$ and $tsk(tid, \_, \_, s) \in \mathcal{Q}$ such that $pp(s) = p$.*

We sometimes say that p is active in task S without referring to the corresponding task identifier. Intuitively, this means that there is a task in S whose next instruction to be executed is the one at program point p.

Definition 2. *Given a program P, its MHP is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S | S_0 \leadsto^* S\}$ where $\mathcal{E}_S = \{(p_1, p_2) \mid p_1$ is active in S within tid_1, p_2 is active in S within tid_2 and $tid_1 \neq tid_2\}$.*

The above definition considers the union of the pairs obtained from all derivations from S_0. This is because execution is non-deterministic in two dimensions: (1) in the selection of the object that is chosen for execution, different behaviours (and thus MHP pairs) can be obtained depending on the execution order, and (2) when there is more than one task, the selection is non-deterministic.

The MHP pairs can originate from *direct* or *indirect* task creation relationships. For instance, in the program of Fig. 2, the parallelism between the points of the tasks executing init and go is *indirect* because they do not invoke one to the other directly, but a third task main invokes both of them. However, the parallelism between the points of the task go and those of incrg is *direct* because the first one invokes directly the latter one. Def. 2 captures both forms of direct and indirect parallelism and they are indistinguishable in the MHP pairs.

3.2 Static Analysis

In [3], an MHP static analysis is presented. Intuitively, the analysis is formalized as a two-phase process: (1) First, methods are locally analyzed and the analysis learns about the tasks invoked within each method, the instructions **await**, **get** and **return** in the following way. A method call creates a task that may run in parallel with the current task. For instance, the task created at 3 runs in parallel with the current one. If the object in which the task is posted is **this** (e.g., when we invoke incrf ()) then the task is pending in the object **this** since we know that it cannot start to execute until a release point is reached in the current task. From **await** and **get** used with futures, the analysis learns that the corresponding tasks have finished after such instructions execute. For instance, at 28, we know that the task created at 27 is finished. Also, at 13 we know that the task created at 12 is finished. This is an essential piece of information for precision. Also, tasks that were pending in this object may become active at the **await** point. For instance, the call considered pending before incrf () would be considered as potentially active at 13, since the processor might be released at this point (and indeed it will necessarily as y cannot be ready until incrf terminates). Likewise, from **return**, we learn that tasks that were pending might become active, as the processor is released by the current task. (2) Second, the information gathered at the local phase is composed by means of an MHP-graph, denoted $\mathcal{G}_P$, which contains all MHP information for a program P. The graph allows us to obtain the (abstract) set of MHP pairs $\tilde{\mathcal{E}}_P$, which are an over-approximation of $\mathcal{E}_P$. Informally, the set $\tilde{\mathcal{E}}_P$ is obtained by checking certain reachability conditions on $\mathcal{G}_P$. Such conditions are not relevant to the contents of this paper (see [3]).

Theorem 1 (soundness [3]). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

3.3 Naïve Extension

The analysis in [3] only handles synchronization instructions, "**await** y?" and "y.**get**", that allow synchronizing one task with the completion of other tasks. Handling "**await** b", where b is a Boolean condition, poses new challenges on the MHP analysis since, unlike "**await** y?", it is not immediate to know which part of the program it synchronizes with, i.e., which instructions make this condition satisfiable. The analysis in [3] can be naïvely extended to soundly, but not precisely, handle this new instruction. This is simply done by treating it as "**await** z?" where z is a fresh variable that does not appear anywhere in the program. This allows pending tasks to become active after the **await**, and thus guarantees soundness. However, it will not mark any task as finished as z is not associated to any instruction or task, and thus it leads to imprecise results.

Example 2. Consider the program depicted in Fig. 2, and note that program point 25 synchronizes with program point 16 using a Boolean condition. Applying the basic MHP analysis with the above naïve extension, we will conclude that $(14, 26)$ and $(29, 11)$ are MHP pairs. These are spurious pairs since go will not proceed to program point 26 until program point 16 has been executed, i.e., go waits for init to finish at program point 16. In practice, these spurious MHP pairs lead to imprecision in analyses for more complex properties that rely on this MHP information. This is the case of the termination and resource analyses of [4] which reports a false alarm. Non-termination (and resource unboundedness) is actually feasible if we omit the synchronization condition at line 16, and is obtained as follows: (1) in main we invoke init and go; (2) go starts to execute and calls incrg that executes and increases the counter g by one; (3) at the release point in line 28, task init starts to execute and invokes incrf (which increases the counter of the loop at go by multiplying it by two); (4) thus, the execution of both loops interleaves and the tasks invoked within the loops modify the counter of the other loop such that the interleaved execution will not terminate; In the presence of the condition at line 25 this termination and resource boundness problem becomes infeasible, since the loop of go cannot start until init has finished and all instances of the tasks incrf that modify the loop counter and thus finished as well. It will be eliminated later when the MHP analysis is enhanced to infer that program points (14 and 26), and also (29 and 11) cannot execute in parallel.

4 MHP Analysis with Condition Synchronization

The goal of this section is to leverage the MHP analysis described in Sec. 3 to handle condition synchronization. For this, we assume that the basic MHP analysis (with the näive extension of Sec. 3.3) has been applied, and the result is given as a set of MHP pairs $\tilde{\mathcal{E}}_P$. Then, we present an extension that eliminates infeasible MHP pairs, as the one of Ex. 2, from this set. This extension is based on a property that we call *must-have-finished* that can be directly used to eliminate infeasible MHP pairs.

The rest of this section is organized as follows: Sec. 4.1 defines the *must-have-finished* property and discusses a specific way to under-approximate it; Secs. 4.2-4.4 describe a practical way to carry out this under-approximation; and finally Sec. 4.5 sketches some important improvements to the overall analysis.

4.1 Must-Have-Finished Property

The *must-have-finished* (MHF) property is defined by means of MHF-sets that are assigned to each program point. Intuitively, the MHF-set for a given program point p is a set of program points that will never execute after p has been reached. Next we formally define the MHF property.

Definition 3. *Given two program points p_1 and p_2, we say that p_2 is in the MHF-set of p_1, denoted $p_2 \in \mathtt{mhfset}(p_1)$, iff for any trace $t \equiv S_0 \leadsto^* S \leadsto^* S'$, if p_1 is active in S, then p_2 is not active in S'.*

Intuitively, this definition states that whenever p_1 is reached (in state S), then p_2 is not active from that state on (note that S' could be equal to S). Clearly, if $p_2 \in \mathtt{mhfset}(p_1)$ then p_1 and p_2 cannot execute in parallel.

Corollary 1. *If $p_2 \in \mathtt{mhfset}(p_1)$, then $(p_1, p_2) \notin \mathcal{E}_P$.*

Example 3. For the program of Fig. 2, we have $16 \in \mathtt{mhfset}(26)$. This is because for method go to proceed to program point 26, the condition at program point 25 must be satisfied. The instruction at program point 16 is the only one that makes this condition satisfiable and it is executed only once. Therefore, it will not execute again after 26 has been executed. This will be used in Sec. 4.5 to guarantee that the method init, together with all instances of method incrg are finished at program point 26.

MHF is a non-trivial undecidable property, and thus we aim at computing under-approximations of it, that can be soundly used in Cor. 1, as follows: in order to include p_2 in the MHF-set of p_1 we require that (1) p_2 *must-have-happened* (MHH) before p_1, i.e., whenever p_1 is reached, then p_2 must have been executed in a previous transition at least once; and (2) p_2 is *unique*, i.e., p_2 executes at most once in any trace. These two properties together imply $p_2 \in \mathtt{mhfset}(p_1)$. Next we formally define the notion of MHH-sets and the set of unique program points, and then explain how they are combined to under-approximate the MHF-sets.

Given a trace $t \equiv S_0 \leadsto^* S_n$, we say that program point p is *executed* in step $i < n$ of t, iff p is active in S_i within task tid and it is not active in S_{i+1} within task tid (note that p could be active in S_{i+1} within $tid' \neq tid$).

Definition 4. *We say that p_2 is in the MHH-set of p_1, denoted $p_2 \in \texttt{mhhset}(p_1)$, iff for any trace $t \equiv S_0 \leadsto^* S_n$, if p_1 is active in S_n then p_2 is executed at least once in step $i < n$ of t.*

Given a trace $t \equiv S_0 \leadsto^* S_n$, we say that a program point p is *reached* in state S_i, with $0 \leq i \leq n$, iff p is active in S_i within task tid, and, either $i = 0$ or p is not active in S_{i-1} within task tid. This means that task tid has reached program point p due to the execution step $S_{i-1} \leadsto S_i$, or, when $i = 0$, p is active in the initial state S_0 within task tid.

Definition 5. *A program point p is unique iff in any trace it is reached at most once. The set of unique program points is denoted by $\mathcal{U}$.*

Lemma 1. $\texttt{mhhset}(p) \cap \mathcal{U} \subseteq \texttt{mhfset}(p)$

Unfortunately the MHH and uniqueness properties are still non-trivial. In Secs. 4.2 and 4.3 we describe an analysis for under-approximating the MHH-sets, and in Sec. 4.4 we explain how we under-approximate the set $\mathcal{U}$.

It is worth noting that Def. 3 and the corresponding under-approximation, capture behaviours in which one part of the program synchronizes with some initialization tasks (since we forbid p_2 to execute again after p_1). This might seem restrictive since it is common to have several instances of the same initialization method, for different objects, executing independently at different times. However, in practice, our analysis uses object-sensitive information [14, 5] and thus considers instances of the same method, when they run on different objects, as if they were different methods. The above formalism can be trivially adapted to an object-sensitive setting, but for the sake of simplifying the presentation, we keep it object-insensitive while our implementation is object-sensitive.

Example 4. Consider the program of Fig. 2, and suppose that the code of main is duplicated, e.g., add "A b = **new** A(); b!init(); b!go();" immediately after line 4. In such case, program point 16 must execute at least once before program point 26. However, since there are two instances of init, one will have to execute before program point 26, but the other might start executing after one of the instances of method go has reached program point 26. Ignoring the object-sensitive information we would conclude that $16 \notin \texttt{mhfset}(26)$. This is indeed correct if a and b were pointing to the same objects, which is not the case.

By taking object-sensitive information into account, our implementation considers the two calls to init (resp. go) as calls to different methods, and thus it is able to prove that $16 \in \texttt{mhfset}(26)$ for objects a and b.

4.2 Under-approximating MHH-sets

The analysis for inferring MHH-sets consists of the following two steps: in the first one we extract what we call MHH-seeds, which are under-approximations of the MHH-sets of some program points that are relatively easy to compute; and then, in a second step, these seeds are propagated to other program points. In this section we detail the second step, while for the first step we just define the notion of MHH-seeds and leave the automatic inference details to Sec. 4.3.

Formally, an MHH-*seed* for a program point p, denoted $\texttt{mhhseed}(p)$ is a subset of $\texttt{mhhset}(p)$. For now, we assume that every program point is assigned an MHH-seed. In general, for most program points it is the empty set. These sets can be provided by the user, or automatically inferred as discussed in Sec. 4.3.

Example 5. For the program in Fig. 2, we let: $\texttt{mhhseed}(26) = \{16\}$, due to the synchronization of program points 16 and 25; $\texttt{mhhseed}(14) = \{19\}$, because program point 13 waits for incrf to finish; and $\texttt{mhhseed}(29) = \{22\}$ since program point 28 waits for incrg to finish. For any other program points p we let $\texttt{mhhseed}(p) = \emptyset$.

Next we describe how to propagate the MHH-seeds to other program points. This is done using a *must* data-flow analysis [17] which merges MHH-sets using *set intersection.* Let $\texttt{pre}(p)$ be the set of program points that immediately precede program point p. When p is an entry program point of method m, then $\texttt{pre}(p)$ consists of the program points at which m is invoked.

Definition 6. *Given a program P, its system of MHH equations E_P includes an equation $E_p = E_p^1 \cup E_p^2$, for each program point p, where $E_p^1 = \cap\{E_{p'} \cup \{p'\} \mid p' \in \texttt{pre}(p)\}$ and $E_p^2 = \cup\{E_{p'} \cup \{p'\} \mid p' \in \texttt{mhhseed}(p)\}$*

Note that E_p^1 takes the intersection of the MHH-sets of all program points that reach p in one step. This is because it is a *must* analysis and thus we should take the information that holds on all paths that lead to p. On the other hand, E_p^2 takes the union of the MHH-sets of those program points that are guaranteed to have been executed before executing p (i.e., its MHH-seeds). Computing the MHH-sets amounts to finding a solution for the equations. We use $\tilde{E}_P$ (and $\tilde{E}_p$ for a program point p) to denote such a solution. Note that these set of equations is object-sensitive since our analysis uses object-sensitive information [14, 5].

Example 6. The following is the set of MHH equations for the program of Fig. 2:

$$\begin{array}{l|l|l}
E_2 = \emptyset & E_{13} = E_{12} \cup \{12\} & E_{25} = E_4 \cup \{4\} \\
E_3 = E_2 \cup \{2\} & E_{14} = E_{13} \cup \{12\} & E_{26} = ((E_{25} \cup \{25\}) \cap (E_{29} \cup \{29\})) \\
E_4 = E_3 \cup \{3\} & \quad \cup E_{19} \cup \{19\} & \quad \cup E_{16} \cup \{16\} \\
E_{11} = (E_3 \cup \{3\}) & E_{16} = E_{11} \cup \{11\} & E_{27} = E_{26} \cup \{26\} \\
\quad \cap (E_{14} \cup \{14\}) & E_{19} = E_{12} \cup \{12\} & E_{28} = E_{27} \cup \{27\} \\
E_{12} = E_{11} \cup \{11\} & E_{22} = E_{27} \cup \{27\} & E_{29} = E_{28} \cup \{28\} \cup E_{22} \cup \{22\}
\end{array}$$

Note that equations E_{14}, E_{26} and E_{29} merge all information from the program points in their MHH-seeds. Solving the above equations we get, among others, $16 \in \tilde{E}_{28}$. Thus, program point 16 must execute at least once before 28.

Lemma 2. *For a program point p, we have $\tilde{E}_p \subseteq \texttt{mhhset}(p)$.*

4.3 Automatic Inference of MHH-Seeds

In this section we explain how we automatically infer the MHH-seeds that are required in Sec. 4.2. In particular we explain how our implementation extracts MHH-seeds from "**await** y?" (or equivalently "y.**get**") and "**await** b" instructions. For the sake of simplifying the presentation, we assume that **await** instructions do not appear at the end of a conditional branch or a loop body; otherwise, we can add a special **skip** instruction to avoid such cases. Moreover, we assume that each method has a single **return** instruction. Recall that to include p_2 in mhhseed(p_1) we should guarantee that p_2 always executes before p_1 at least once.

Extracting MHH-seeds from "await y?" instructions. Given an "**await** y?" instruction at program point p, if it is guaranteed that y refers to only one method m, then we add the program point of the **return** instruction of m, to the MHH-set of the program point p' that immediately follows p. In practice, we identify the method to which y refers using points-to analysis as in [9].

Example 7. Consider the program of Fig. 2. Applying this technique to the **await** instruction at program point 28, we add program point 22 to mhhseed(29) since the future variable z refers to method incrg.

Extracting MHH-seeds from "await b" instructions. Given an "**await** b" instruction at program point p, we first syntactically look for instructions that make b satisfiable. This search is based on some heuristics depending on the shape of the condition b. For example, for the condition at program point 25 (in Fig. 2), we look for instructions that assign a non-**null** value to x. In addition, we require that there is only one such instruction, and that no other instruction in the program can make b unsatisfiable – the only allowed way to be unsatisfiable is by the default values assigned to the variables when creating the corresponding objects. This condition can be refined further, such that when adding p' to mhhseed(p), we also add any program point that follows p' until we reach an **await** instruction. This is because after p' is executed, the processor will not be released until the next **await** instruction.

4.4 Under-Approximating the Set $\mathcal{U}$ of Unique Program Points

A sufficient condition for a program point p to be unique is that it belongs to a method m that is unique (i.e., m is executed only once) and that it is not part of a loop. The challenge is how to prove that a given method m is unique.

Recall that in the semantics of Sec. 2.2 tasks never disappear once finished, but they rather remain at their exit point during the remaining execution. This behavior is correctly handled by the basic MHP analysis. Consequently, if a method m executes more than once, eventually some of its program points will "run" in parallel. This is true even if these instances of m execute one after the completion of the other. Therefore, if there are no program points p_1 and p_2 in m such that $(p_1, p_2) \in \tilde{\mathcal{E}}_P$ (the result of the basic MHP analysis that we have assumed to be available), we can safely conclude that m is unique.

Example 8. We have seen in Ex. 6 that $16 \in \tilde{E}_{28}$, i.e., program point 16 is in the MHH-set of program point 28. Since 16 does not appear in a loop and method init is unique, we can conclude that 16 is in the MHF-set of 28.

4.5 Further Improvements

Eliminating MHP pairs using the unique methods structure. We can extend the computed MHF sets by analyzing the structure of unique methods. In the example of Fig.2 , we cannot conclude that $14 \in \texttt{mhfset}(28)$ since 14 is not unique. Moreover, $14 \notin \texttt{mhhset}(28)$ because the body of the loop might not be executed at all. Fortunately, 16 still belongs to $\texttt{mhfset}(28)$ and a syntactic analysis of method init allows us to conclude that $14 \in \texttt{mhfset}(16)$. Given that 16 is guaranteed to execute before 28, we can conclude that $14 \in \texttt{mhfset}(28)$.

Eliminating MHP pairs using the MHP graph $\mathcal{G}_P$. In the previous extension, we were able to discard all the program points of init (with respect to 28) but we cannot discard 19 because incrf is not unique. However, all instances of incrf are called from init and executed while init is in the loop, before 16. This fact is reflected in the MHP graph $\mathcal{G}_P$ (see Sec. 3). All the paths that allow us to reach 19 from 28 traverse one of the program points of init which are guaranteed to have finished by the time 28 is reached. In fact, 19 and 28 cannot happen in parallel. This can be detected by incorporating the MHF-sets in the corresponding MHP-graphs $\mathcal{G}_P$. Given a program point p its corresponding set of refined MHP pairs, denoted $\tilde{\mathcal{E}}_P^p$, is constructed as follows: (1) first remove all nodes that correspond to program points in $\texttt{mhfset}(p)$ from the MHP graph $\mathcal{G}_P$; and (2) construct a set of MHP pairs from the modified graph using the same reachability conditions as in [3]. Then, given two program points p_1 and p_2, if $(p_1, p_2) \notin \tilde{\mathcal{E}}_P^{p_1} \cap \tilde{\mathcal{E}}_P^{p_2}$, the pair (p_1, p_2) can be safely eliminated from $\tilde{\mathcal{E}}_P$. With this improvement, we can guarantee that the method incrf will not be executed during the await 28. Therefore, we will be able to prove termination of the while loop in method go.

5 Conclusions and Related Work

Static analysis of concurrent programs is considered complex and computationally expensive. The analysis is expensive because one needs to consider all possible interleavings of processes. Cooperative scheduling, as used in our concurrency model, greatly alleviates this problem because interleavings can only occur at explicit points (in contrast to preemptive scheduling in which interleavings must be considered at every point). As regards complexity of the analysis, one of the main challenges is on understanding the synchronization of processes. In the context of cooperative scheduling, the problem was partially solved in [3], where synchronization based on future variables was accurately treated. This work extends such previous analysis to handle synchronization on shared variables effectively, thus enabling the analysis of programs that use both mechanisms.

The MHP analyses of [13, 2] for X10 focus on synchronization idioms that do not allow conditional synchronizations. In Java, condition synchronization can be simulated using `wait-notify`. The analysis of [16] supports `wait-notify`. However, in this context the relation between the waiting and notifying threads is easier to identify (in our case this is done by the seeds). The analysis of [6] does not support `wait-notify`. The analysis of [15] does not support any synchronization idiom of Java, it just analyzes the thread structure of the program.

References

1. Ericsson AB. *Erlang Efficiency Guide*, 5.8.5 edition, October 2011. From `http://www.erlang.org/doc/efficiency_guide/users_guide.html`.
2. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proc. of PPOPP'07*, pages 183–193. ACM, 2007.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51.
4. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*, Lecture Notes in Computer Science. Springer, October 2013. To appear.
5. E. Albert, P.Arenas, J. Correas, M. Gómez-Zamalloa, S. Genaim, G. Puebla, and G. Román-Díez. Object-sensitive cost analysis for concurrect objects. http://costa.ls.fi.upm.es/papers/costa/AlbertACGGPRtr.pdf, 2012.
6. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer, 2005.
7. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
8. M. Emmi, A. Lal, and S. Qadeer. Asynchronous programs with prioritized task-buffers. In *SIGSOFT FSE*, page 48. ACM, 2012.
9. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, Lecture Notes in Computer Science, pages 273–288. Springer, 2013.
10. E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
13. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Proc. of PPoPP'10*, pages 25–36. ACM, 2010.
14. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
15. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
16. G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. *SIGSOFT Softw. Eng. Notes*, 24(6):338–354, 1999. 319252.
17. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 2005. Second Ed.